Understanding SVA Degeneracy (Revised)

Ben Cohen

11/19/2023

This paper provides an explanation of SystemVerilog "degenerate sequences" and their associated restrictions. Additionally, it explores the concept of an "empty match" and the rules that govern the integration of empty matches into sequences. A case study involving the **within** operator is included to illustrate potential issues that may arise when used with an empty match.

Notes:

- `[*0` in red indicates caution in use, it may be problematic in certain circumstances, as explained in the paper.
  `[*0` in blue indicates legal.

1.1 What is a sequence, a match, a no match, an empty match?

1.1.1 Definition

1800 defines a simple sequence as a finite list of SystemVerilog Boolean expressions in a linear order of increasing time. A sequence is multi-threaded if it includes any of the following operators: **and,intersect,or**, range delay (e.g., `##[1:5]`), consecutive range repetition (e.g., `a[*1:5]`), repetitions with the goto (e.g., `a[->1],` and non-consecutive repetition operators (e.g., `a[=2]`). Essentially, regular expressions constitute the foundation of sequences. Examples of sequences using the variables `a,b,c` are: `(a ##1 b);` *// a single-threaded linearsequence*

`(a ##1 b) or (a ##[1:$] c[*2]);` *// a multi-threaded sequence*

1.1.2 Outcomes of a Sequence

Sequences can have various outcomes, including MATCH, NOT MATCHED, NO MATCH (*never a match*), or EMPTY MATCH. A sequence can also be considered DEGENERATE or NONDEGENERATE, as explained below. The endpoint of a sequence is either a match or not a match. The endpoint is converted to logic values using the sequence methods **triggered** and **matched** or when casted as a property.

A sequence is considered MATCHED when, upon examining its construction, the elements of its threads are such that the sequence expression is satisfied. Conversely, a NOT MATCHED outcome occurs otherwise. For example:

```
(a ##1 b);  // is a match when a==1 and b==1 in the next cycle
(a ##1 b) or (a ##[1:$] c[*2]);  // is a match when a==1 and b==1 in the next cycle
          // OR  when a==1 and c is repeated twice at some future cycles.
(a ##1 b)  and (a ##[1:$] c[*2]);  // is a match when a==1 and b==1 in the next cycle
          // AND  when a==1 and c is repeated twice at some future cycles.
```

It is possible to have a sequence that admits NO MATCH, meaning that, by construction, the sequence can never match. For example, `(a ##1 1'b0 ##1 b), ((a ##1 b) and (a ##[1:$] 1'b0))` are sequences that can never match because of the hard `1'b0`. Sections 1.2.1 rule 1 and 1.2.1 rule 2 define two other conditions that are NO MATCH: `(a[*0] ##0 seq)` and `(seq ##0 a[*0])`. Note that there is a distinction between a sequence that allows for NO MATCH and a sequence that is not matched. The former has no possibility of a match, while the latter has such a possibility depending on the values of the variables at sampled times.

It is also possible for a sequence to admit an EMPTY match using the `[*0` repeat operator, such as `a[*0]`, `a[*0:2]`. Both of these examples admit an empty match, the `a[*0]` element of a sequence cannot be sampled. A sequence may admit both empty and nonempty matches, for example, `a[*0:2]` admits an empty match and up to two nonempty matches, `a[*1]` and `a[*2]`.

Broadly speaking, a DEGENERATE sequence is a sequence created in such a way that it lacks potential matches, leading to incoherent and meaningless assertions. The following sections addresses degeneracy and the results of concatenating empty sequences with other sequences.

1.2 DEGENERATE SEQUENCES

IEEE 1800'16.9.2.1 Repetition, concatenation, and empty matches in combination with *1800'16.12.22 Nondegeneracy* define the criteria for identifying degeneracy and its restrictions. The following table defines when a sequence is degenerate.

RULE	COMMENTS
1. A sequence that admits **NO MATCH** or that admits **only empty matches** is called **degenerate**.	A sequence is a NO MATCH if by construction it can never match, see discussion in 1.1.2. `a intersect (b ##1 c)` *// Admits no match because sequence (a)* *// of length 1 can never intersect sequence (b ##1 c) of length 2.* `a[*0]` *// admits only empty match. a[*0] is not sampled and it* *// is not concatenated with another sequence as per 1.2.1 rules 2, 3* *//* *// Both of the above examples are degenerate*
2. A sequence that admits at least one nonempty match is called **nondegenerate**. The following restrictions apply:	
a. Any sequence that is used as a **property** shall be **nondegenerate** and **shall not admit any empty match.**	`b |-> a[*0:2];` *// Illegal because it admits an empty match, the* `[*0]` `b |-> a and (b ##1 0);` *// Illegal and degenerate because* `// (b ##1 0)` *is a NO MATCH per 1.2 rule 1*
b. Any sequence that is used as the **antecedent** of an overlapping implication **(|->)** shall be **nondegenerate**.	`a[*0] |-> b;` *// Illegal and degenerate because,* *// it admits an empty match the* `a[*0]` `a and (b ##1 0) |-> b;` *// Illegal and degenerate because* `// (b ##1 0)` *is a NO MATCH per 1.2 rule 1*
c. Any sequence that is used as the antecedent of a nonoverlapping implication **(|=>) shall admit at least one match.** Such a sequence can admit only empty matches.	`a[*0] |=> b` *// is legal because* `|=>` is same as `##1 1 |->` `a[*0] |=> b` *// is equivalent to:* `empty ##1 1 |-> b` *// Also, per empty rule 1.2.1 rule 3* *// where* `(empty ##1 1)` *is same as* `##0 1,` *the property is same as* `##0 1 |-> b` *// nondegenerate* `a[*0:2]|=>` *// admits at least 1 match. It is same as* `(##0 1) or (a[*1] or a[*2]) |-> b` *// nondegenerate*

1.2.1 Empty match rules

`a[*0]` is an empty match and it follows the set of concatenation rules specified below. In the following rules, an empty sequence is denoted as *empty*, and another sequence (which may be empty or nonempty) is denoted as *seq*.

	RULES AND EXAMPLES	COMMENTS
1	**(empty ##0 seq) does not result in a match** `a[*0] ##0 b |-> c;  c |-> a[->0] ##0 b;` *// see section 1.3 on [->0], [=0]* `———————— Analysis ———————` `(a[*0] ##0 b) |=> c;` *// is same as* `(a[*0] ##0 b) ##1 1 |-> c;` *// is same as* *// The ##n associates to the left, thus* `( NO_MATCH ) ##1 1 |-> c;` *// same as* `( 0 ) ##1 1 |-> c;` *(degenerate, NO MATCH)*	Since (`empty ##0 seq`) is a NO MATCH. 1.2 rules 2a, 2b (above) make this sequence degenerate and thus illegal regardless of where it is used, a property or as an antecedent —————— Note: `(NO_MATCH ##1 1)` is **NOT** the same as `(empty ##n seq)` (rule 3 below) `(NO_MATCH ##1 1)` is a sequence that admits no match, meaning it can never have a match; thus, that sequence is degenerate and illegal per 1.2 rule 2c which requires an admission of at least one match.
2	**(seq ##0 empty) does not result in a match** `b ##0 a[*0] |-> c;` `c |-> b ##0 a[->0];` `c |-> b ##0 a[=0];` `————————` `(b ##0 a[*0]) |=> c;` // same as `(b ##0 a[*0]) ##1 1 |-> c;` // same as `( NO_MATCH ) ##1 1 |-> c;` `( 0 ) ##1 1 |-> c;` *// Can never matched*	Since (`seq ##0 empty`) is a no match, 1.2 rules 2a, 2b (above) make this sequence degenerate and thus illegal regardless of where it is used, a property or as an antecedent
3	**(empty ##n seq), where n is greater than 0, is equivalent to (##(n-1) seq)** `(a[*0] ##2 b) |-> c;` *// same as* `( ##1 b) |-> c;` `------------------` `(a[*0:$] ##2 b |-> c;` *// same as* `(a[*0] ##2 b) or (a[*1:$] ##2 b) |-> c;` *// same as* `##1 b) or (a[*1:$] ##2 b) |-> c;` `----------------------------` `a[*0] |=> c;` *// same as* `[*0] ##1 1 |-> c;` *// same as* `##0 1 |-> c;`	(`empty ##n seq`) is equivalent to (`##n-1 seq`) thus it is nondegerate because there can be a match if *seq* matches; it is <u>not</u> a NO MATCH.
4	**(seq ##n empty), with n > 0, is equivalent to (seq ##(n-1) `true)**  `w ##1 b[*0] |-> c;` *// same as* `w ##0 1 |-> c;` `-----------------` `(w ##1 b[*0:$] ##1 d) |-> c;` *// same as* `(w ##1 b[*0] ##1 d) or` `(w ##1 b[*1:$] ##1 d) |-> c;` *// same as* `-------------` `(w ##0 1 ##1 d) or` `(w ##1 b[*1:$] ##1 d) |-> c;`	Note: `` `true `` is same as `1'b1` (`seq ##n empty`) is equivalent to (`seq ##n-1 1`) thus it is nondegerate because there can be a match if *seq* matches; it is <u>not</u> a NO MATCH.

1.3 Why are the [->0] and the [=0] repetitions empty matches ?

Why are the goto [->0] and the Nonconsecutive [=0] repetitions empty matches? To understand why. we need to understand the equivalency of these operators.

`b[->n]` *is equivalent to*

```
   ( OR[i in 0:$](!b[*i] ##1 b) )[*n]
```

for n==0, we get

```
   ((!b[*0] ##1 b) or (!b[*1] ##1 b) or (!b[*2] ##1 b) or ...          )[*0]
   ( <----------------- a_sequence ------------------------------->)[*0]
```

Thus **`b[->0] is equivalent to ( a_sequence)[*0], an empty match`**

Similarly, per 1800 16.9.2, `b[=n]` is equivalent to `b[->n] ##1 !b[*0:$]`

`for n==0,` we get `(b[->0] ##1 !b[*0:$])`

`Thus b[=0]` is equivalent to `(b[*0] ##1 !b[*0:$]),` equivalent to

`(b[*0] ##1 !b[*0])` **`or`** `(b[*0] ##1 !b[*1:$]),` equivalent to

`(##0 !b[*0]) or (##0 !b[*1:$])`

Revised

1.4 Why can the `within` operator be problematic?

The **`within`** sequence operator can result in unexpected outcomes when the final term of the contained sequence has an empty element. To comprehend the underlying cause, we must examine the definitions of the **`within`** operator. Specifically,
`(seq1 within seq2)` is equivalent to:

```
    ((1[*0:$] ##1 seq1 ##1 1[*0:$]) intersect seq2 )
```

An issue occurs when `seq1` has an element that ends with a [*0, or a [*0:w] (*w is a number*). For example,
If `seq1` is a sequence of the form `(seqA ##k seqB[*0:n]);` then for `n==1,` the above becomes
`((1[*0:$] ##1 seqA ##k seqB[*0:1] ##1 1[*0:$]) intersect seq2 ),` where `k is >= 0`
This is equivalent to

```
((1[*0:$] ##1  ( seqA ##k seqB[*0]    ##1  1[*0:$]) intersect seq2 )  or  // thread1
((1[*0:$] ##1  ( seqA ##k seqB[*1:1]) ##1 1[*0:$]) intersect seq2 )  // the other threads
```

Using left associativity of the `##k` operator, *thread1* can be reduced to the following when we use this empty rule:

```
(seq ##n empty)  is equivalent to (seq ##(n-1) `true)  // with n>0
((1[*0:$] ##1  ( seqA ##k-1 `true      ##1  1[*0:$]) intersect seq2 )  // thread1
```

The whole sequence is then reduced to:

```
((1[*0:$] ##1  ( seqA ##k-1 `true      ##1  1[*0:$]) intersect seq2 ) or  // thread1
((1[*0:$] ##1  ( seqA ##k seqB[*1:1])  ##1  1[*0:$]) intersect seq2 )  // the other threads
```

Note that *thread1* does not include an instance of **`seqB.`** For the entire sequence to match, it is only necessary for a thread within *thread1* with a matching **`seqA`** to intersect with **`seq2`**. The other **`OR`**ed threads are not required for evaluating this sequence since **`(seqB[*1:1]) ##1 1[*0:$])`** from the other threads is satisfied with **`(``true ##1 1[*0:$])`** in *thread1*. Thus, we can conclude that a sequence of the form **`(seqA ##k seqB[*0:n] within seq2)`** absorbs the **`seqB`** sequence, making it unnecessary. This can lead to unintended results.

1.4.1 User Example of `within`

Below is an actual user case where the requirements stated that between two rises of `req` starting from the next cycle, **there should be one and only one new `gnt`.** Intuitively, or perhaps naively, one would write something like the following property: `$rose(req) |=> $rose(gnt)[=1] within $rose(req)[->1]`

Simulation of the above property showed that if failed to throw an assertion error when 2 or more new occurrences of `gnt` occurred. But why? The culprit is the Nonconsecutive repetition operator `[=` in combination with the **`within`** operator.
Specifically, as mentioned above
1) `(seq1 within seq2)` is equivalent to: `((1[*0:$] ##1 seq1 ##1 1[*0:$]) intersect seq2 )`
2) `seq1` is defined here as `($rose(gnt)[=1])` and it is equivalent to
`!$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0:$]`
Expanding the consequent property:
`($rose(gnt)[=1]) within $rose(req)[->1])` *// becomes*

```
(1[*0:$] ##1
    !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0:$]
        ##1 1[*0:$])  intersect $rose(req)[->1])
```

Consider the LHS of the intersect,
`(1[*0:$] ##1 !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0:$] ##1 1[*0:$])`
It gets simplified to:
`(1[*0:$] ##1 !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0] ##1 1[*0:$]) or` *// **thread1***
`(1[*0:$] ##1 !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*1:$] ##1 1[*0:$])` *// other threads*

We don't need the other threads because ***thread1*** of the ORed sequences is sufficient here.
Due to the empty rule: *`(seq ##n empty)` is equivalent to `(seq ##(n-1) `true)`* ***thread1*** gets reduced from:
`(1[*0:$] ##1 !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0] ##1 1[*0:$])`
To

```
(1[*0:$] ##1 !$rose(gnt)[*0:$] ##1 $rose(gnt) ##0       `true      ##1 1[*0:$])
```

Thus, the assertion does not test for a 2nd occurrence of `$rose(gnt)`. A 2nd occurrence of `$rose(gnt)` does not cause a failure. For a pass, all we need here with the **`within`** is a single occurrence of `$rose(gnt).`

This property can be correctly written as:
$rose**(req) |=> **$rose(gnt)[=1] **intersect $rose**(req)[->1]

Expanded the consequent property, we get

```
!$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0:$] // equivalent to
      !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*0]  or  // thread 1, same as
 //   !$rose(gnt)[*0:$] ##1 $rose(gnt) ##0    1             or  // Thread1 per empty match rules
      !$rose(gnt)[*0:$] ##1 $rose(gnt) ##1 !$rose(gnt)[*1]  or  // thread 2
       … and so on
```

If any of the threads lengthwise-intersect the 2nd occurrence of `$rose(req)`, then in that intersected thread there can only be one occurrence `$rose(gnt)`. The user code and results are available at
https://www.edaplayground.com/x/DwJ2 // code
https://www.edaplayground.com/w/x/Ueg // wave

1.5 How tools handle degeneracy

As discussed in Section 1.2, 1800 establishes rules, specifying, for instance, that any sequence utilized as a property must be nondegenerate and must not allow any empty match. However, 1800 lacks clarification on how tools should manage these conditions or offer guidance on potential actions; 1800 is a specification, it does not address tools. When encountering degeneracy, should a tool outright reject the compilation? Should it provide a compilation warning and disregard the assertion? Alternatively, should it simply overlook the issue and treat the degenerate sequence as not a match?

There are legitimate cases where specifying a property using a macro or a `generate` parameter can render a property degenerate (and thus useless) for some configurations while remaining meaningful for others. In practical applications, the generation of degenerate and illegal sequences can be subtle for the user, influenced by factors like configuration options or `generate` loop statements. For example, in scenarios like:
`special_ack |-> SPECIAL_RESOURCE_IN_CONFIG`, where the parameter in caps may vary by config, users might not want compilation errors in the cases where the current configuration makes that particular property useless. Another example is with the use of the `generate`, such as:

```
generate
    genvar i, j;
    for (i = 0; i < 2; i++) begin
       for (j = 0; j < 2; j++) begin
        ap_abc: assert property (@ (posedge clk)
          $rose(a) |-> b[*j] ##i c);
       end
    end
endgenerate
// equivalent t0:
ap_abc00: assert property (@ (posedge clk) $rose(a) |-> b[*0] ##0 c); // degenerate
ap_abc01: assert property (@ (posedge clk) $rose(a) |-> b[*1] ##0 c); // legal
ap_abc10: assert property (@ (posedge clk) $rose(a) |-> b[*0] ##1 c); // same as ##0 c
ap_abc11: assert property (@ (posedge clk) $rose(a) |-> b[*1] ##1 c); // legal
```

So how do simulation and formal verification tools handle degeneracy?
I observed that, in simulation, vendors' approaches to handling degeneracy depend on the specific tools and versions. Some tools outright reject compilation when faced with degeneracy, while others may issue a warning or choose to entirely overlook the issue and treat a degenerate sequence as not matched. In the case of a formal verification tool, it may likely allow a degenerate sequence; however, you won't see the property successfully proven or covered.

1.6 Conclusions and Recommendations

Empty and NO MATCH sequences can generate degenerate sequences. The purpose of these rules and restrictions on these types of matches is to prevent the creation of assertions that might result in sequences without potential matches, leading to incoherent and pointless assertions. It is advisable to steer clear of degenerate sequences for clarity and for reasons of tool compatibility, especially if your company has access to multiple vendors.

Exercise caution when using repetition operators that include a zero option, like:
[*0], [*0:n], [->0], [->0:n], [=0], [=0:n], seq1 ##n seq2[*0:k] within seq3, and avoid sequences that, by design, cannot possibly yield a match *(e.g., the use of the* `intersect` *with no potential for a length match between the LHS and the RHS, a sequence with hard* `0s`, e.g., `a ##1 0 ##1 b`).

1.7 Sequence Rule

SVA expression is evaluated from left to right; however, the evaluation must also abode to 1800'2017 sequence evaluation rules, as described below. This is important, particularly when the empty match is involved.

1800: 16.9.11 Composing sequences from simpler subsequences

There are two ways in which a complex sequence can be composed using simpler subsequences. One is to instantiate a named sequence by referencing its name. Evaluation of such a reference requires the named sequence to match starting from the clock tick at which the reference is reached during the evaluation of the enclosing sequence. For example:

```
sequence s; a ##1 b ##1 c; endsequence
sequence rule;
  @(posedge sysclk)
     trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
// Named sequence rule in the preceding example is equivalent to the following:
sequence rule; @(posedge sysclk) trans ##1 start_trans ##1
                    (a ##1 b ##1 c) ##1 end_trans ;
endsequence
```

[comment] Thus, loosely speaking, the **( )** *is like an enclosure that is equivalent to a sequence declaration.*

Sequence **s** is evaluated **beginning one tick after the evaluation of start_trans** in the sequence rule.

1.7.1 Application example with empty matches

Why is this sequence **(req ##1 rdy[*0] ##0 ack)** legal, yet
this sequence **(req ##1(rdy[*0] ##0 ack))** degenerate ?

Answer: Per 1800 sequence rule, **(rdy[*0] ##0 ack)** is evaluated after the evaluation of **req**. Thus,

```
assert property( @(posedge ip_clk)
 req ##1( rdy[*0] ##0 ack ) ); // Is illegal because following the evaluation of req
//        ( rdy[*0] ##0 ack ) //  gets evaluated in the next cycle, and
//        ( rdy[*0] ##0 ack ) //  is degenerate.
```

However,

```
assert property( @(posedge ip_clk)
 req ##1 rdy[*0] ##0 ack); // Is legal because
 // req ##1 rdy[*0] ##0 ack // is reduced to
 // req ##0 1 ##0 ack // using left associativity
```

Acknowledgement: I express my gratitude to Ed Cerny for his advice in reviewing and formulating this paper.

Ben@systemverilog.us
Link to the list of papers and books that I wrote, many are now donated.
http://systemverilog.us/vf/Cohen_Links_to_papers_books.pdf